# 바이트코드 수정을 통한 자바 애플릿보안

박상길[O] 노봉남
전남대학교 전산학과
{sgpark, bongnam}@athena.chonnam.ac.kr

# Java Applet Security by Bytecode Modification

Sang-Kil Park[O] Bong-Nam Noh
Dept. of Computer Science, Chon-Nam National University

**요 약**

자바가상기계(JVM : Java Virtual Machine)는 실행전에 바이트코드를 확인하는 바이트코드 검증기와 실행환경에서 점검하는 바이트코드 인터프리터를 포함한다. 자바 애플릿은 서비스 거부 공격이나, 사용자를 속이기 위한 조작한 링크 정보를 상태바에 보인다거나, 전자메일을 위조하여 보내는 등의 사용자에 유해한 행위를 할 수 있다. 웹브라우저를 통해 유해한 행동을 하는 클래스에 대해 사전에 바이트코드 수정을 통하여 안전한 클래스로 대체한다. 바이트코드 수정에는 클래스 수준 수정과 메소드 수준 수정이 있다. 클래스 수준 수정은 자바의 상속성을 이용하고, final 클래스나 인터페이스처럼 상속되지 않는 클래스는 메소드 수준에서 바이트 코드 수정을 한다. 메소드 수준 수정은 바이트코드 명령과 Constant Pool을 수정한다. 바이트 코드 수정을 적용하면 웹서버, 클라이언트, 브라우저에 대해 어떠한 별도의 작업도 필요없이 프락시 서버에서 유해클래스를 Safe 클래스로 수정한 후 브라우저에 보인다.

## 1. 서론

자바프로그램은 시스템 개발과 웹 페이지의 능동적 요소등의 목적으로 개발된다. 악의적 코드의 실행을 막기 위해 애플릿과 자바프로그램을 실행하기전에 바이트 코드 검증기로 확인하고, 코드의 속성등도 확인한다. 그러나, 이러한 방법도 런타임(run time)시에 바람직하지 못한 서비스거부 공격들을 막지는 못하였다. 따라서 애플릿 속성에 영향을 미치는 메소드(method)등을 사용자의 요청에 용이하게 생성할 수 있는 방법이 필요하며, 바이트코드 수정(bytecode modification)은 런타임 환경에서 애플릿을 바이트코드 수준에서 수정한다. 바이트코드 수정시 추가되는 명령들은 자원의 사용에 대해 감시하고 제어하며, 애플릿의 기능을 제한하고 여러 객체에 대한 접근제어를 제공한다. 바이트코드 수준 수정은 자바언어로만 구현된 *Muffin* 프록시 서버를 사용하며, 바이트코드의 제어를 위해 *JavaClass* 패키지를 설치한 뒤 Safe 클래스를 각 유형별로 추가한다[8].

## 2. 자바 애플릿 보안

시스템에 바람직하지 못한 애플릿의 유형별 공격방법과 그 해결방법을 살펴본다.

### 2.1 서비스 거부 공격

자바가상기계는 서비스 거부 공격에 대한 보호기능은 거의 제공하지 못한다. 애플릿이 CPU 처리시간을 독점하여 시스템을 불안정하게 하거나, 계속된 메모리의 할당으로 시스템을 정지시킬 수 있고, 쓰레드와 시스템 프로세스를 기아상태에 빠지게 할 수 있다[1,6,7]. 실행환경에서 부적절한 시스템 자원의 사용으로부터 자바의 안정성이 위협될 가능성이 있으므로 자원의 사용과 제어에 대한 감시메커니즘이 필요하다.

### 2.2 기밀정보의 유출

웹브라우저들은 애플릿이 로드되는 곳의 유형에 따라 애플릿에게 서로 다른 권한을 부여한다. 애플릿은 사용자의 기밀정보를 URL 리다이렉트등의 위장채널을 사용하여 전송할 수 있으며, 브라우저에게 웹의 특정 페이지를 로드하라고 명령할 수 있다. 가능한 또 다른 위장채널로는 파일에 대한 시간지연 접근, 애플릿간의 정보 공유등이며 이러한 저장매체를 이용한 애플릿 상호간의 통신은 감사장치를 통한 애플릿의 행위를 모니터링을 통해 감지될 수 있다[2].

### 2.3 위장공격

위장 공격에서는 공격자는 사용자가 보안과 관련된 부적절한 결정을 내리도록 속이기 위해 잘못된 정보를 생성한다. 애플릿은 마우스가 그림이나, 링크위를 지날 때 접근가능한 URL을 브라우저의 밑에 있는 상태바에 보여주게 된다. 유해 애플릿이 상태바에 잘못된 URL을 보여주는 것을 눈치채지 못하면 애플릿은 사용자로 하여금 다소 위험한 사이트로 접속하게 만든다. 위장공격의 경우는 상태바에 표현되는 URL에 대한 제어를 통해 해결 가능하다[3].

### 2.4 괴롭힘 공격

자바 애플릿은 시끄러운 소리를 그치지 않고 반복하여 들리게 하여 사용자를 괴롭힐 수 있다. 또 다른 방법은 브라우저가 방문한 사이트를 계속 반복해서 방문하도록

하는 것과, 그때마다 새로운 윈도우를 생성하는 것이다 [5]. 이에 대해서는 웹 브라우저를 종료하여야만 괴롭힘에서 벗어날 수 있다. 웹 브라우저를 종료하지 않고 해결하기위해서는 자바실행환경에서 사운드 객체에 대한 모니터링과 제어가 꼭 필요하다.

### 3. 자바 바이트코드 수정

앞에서 살펴본 공격의 해결책으로 바이트코드 수정 메커니즘을 살펴본다. 기본적 개념은 애플릿 코드에 Safe 코드를 삽입함으로써 제한을 가하는 것이다. Safe코드는 애플릿의 기능을 제한할 뿐만 아니라, 자원의 사용에 대해 모니터링과 제어를 한다. Safe 메커니즘은 실행환경에서 테스트하였을 경우, 클래스 또는 메소드와 같은 실행가능한 요소로서 대치된다. 이러한 Safe 메커니즘은 반드시 애플릿이 실행되기 전에 수행되어 바이트코드를 변경해야만 한다. 그림 1과 같이 웹서버와 웹 브라우저 사이에 Muffin 프록시 서버를 두면 애플릿은 웹 서버와 클라이언트의 브라우저 사이에 위치하는 HTTP 프록시 (Muffin) 서버에 의해 수정된다. 이를 통해 웹서버와 자바가상기계, 웹 브라우저에 대해 어떠한 변화도 필요하지 않고 구현된다는 장점이 있다.

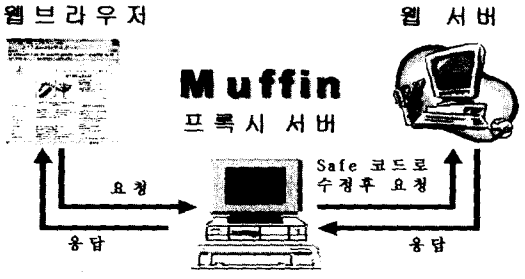


그림 1 Muffin 프록시 서버의 HTTP 처리

애플릿과 브라우저는 다운로드된 애플릿코드의 수정을 알지 못하고, 브라우저의 요구는 프록시 서버를 통하여 Safe코드로 변경된 후 웹 서버에게 요구하게 된다.

### 3.1 클래스 수준 수정(Class-level Modification)

*Window*와 같은 클래스의 경우 탑 레벨 클래스인 *Frame*클래스는 좀 더 제한적인 *Safe$Frame*으로 바뀌어진다. 이는 추가적인 보안과 온전함을 확인하는 루틴을 포함하고 있다. *Safe$Frame* 클래스의 메소드 생성자는 화면에 몇 개까지의 윈도우가 열리도록 제한하며 윈도우의 수가 임계치를 초과하면 새로운 윈도우의 생성은 거부된다. *Safe$Frame*클래스는 *Frame* 클래스의 서브타입이며, *Frame*클래스가 요구되는 모든 곳에 대신하여 존재하게 된다. 애플릿은 임계치이상 윈도우를 생성하기 전까지는 바이트코드 수정을 알 수 없다. 클래스 수준수정은 단순히 *Frame* 클래스에 대한 참조를 *Safe$Frame*으로 바꿈으로서 이루어진다. 자바는 모든 문자열과 클래스, 필드, 메소드들에 대한 참조는 클래스 파일의 Constant Pool을 통해 결정된다. Constant Pool에는 CONSTANT_Class entry를 통해 클래스 이름이 어느 곳에 저장되어 있는지를 나타낸다.

클래스 이름의 CONSTANT_Utf8의 요소 java/awt/

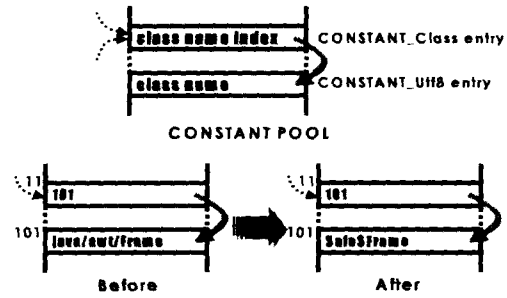


그림 2 클래스 수준의 클래스 수정

*Frame*를 새로운 클래스인 *Safe$Frame*으로 바꾼다면, 그림 2와 같이 CONSTANT_ Class 요소는 *Safe$Frame*클래스로 변경된다. 클래스 수준 수정은 상속성을 이용하여 단순하게 Constant Pool 요소의 수정만을 요구한다. 그러나 클래스의 상속성을 이용할 때 한계에 부딪히는 경우가 있다. *final* 클래스와 인터페이스의 경우에는 상속되지 않으므로, 상속성이 이러한 접근을 금지한다.

### 3.2 메소드 수준 수정(Method-Level Modification)

클래스수준의 수정을 통해 할 수 없는 final 키워드나 인터페이스를 통해 접근 가능한 메소드의 경우 메소드 수준 수정이 사용되며, 좀더 복잡한 메소드 참조와 메소드 호출 명령(method invocation instruction)이 요구된다.

표1의 필드 기술자를 살펴보면 각각은 클래스나 인스턴스 변수의 형식을 표현한다.

표 1) 필드 기술자(descriptor)의 의미

| Descriptor | Type |
|---|---|
| *C* | Character |
| *I* | Integer |
| *Z* | Boolean |
| *L<classname>;* | An instance of the class |

메소드 기술자는 메소드가 갖는 매개변수와 리턴되는 값을 표현한다. 매개변수 기술자는 0 또는 그 이상의 필드 타입들을 갖는다. V 문자는 메소드의 리턴 값이 없을 때 (void)를 나타낸다. 메소드 void ex(Thread t, int i)는 (Ljava/lang/Thread;I)V로 표현된다. 그림 2를 통해 메소드 호출 명령이 어떻게 변경되는지 보기전에, 메소드가 클래스 파일로 어떻게 컴파일 되는지 알아본다.

```
예제 메소드
    void ex(Thread t, int i)
    t.setPriority(i);
컴파일된 메소드
    method public example1(Ljava/lang/Thread;I)V
    push Ljava/lang/Thread;I)V
    push I
    invokevirtual Thread.setPriority(I)V
```

그림 3 예제 메소드와 컴파일된 메소드

클래스 수준 수정은 그림3과 같이Thread.setPriority(I)V를 좀 더 제한적인 메소드인 Safe$Thread.setPriority (Ljava/lang/Thread;I)V로 교체한다. 교체된 Safe$Thread. setPriority 메소드는 applet에게 상위한계치보다 높은 우선순위(priority)를 부여할 수 없도록 한다. ex.setPriority(5)는

Safe$Thread.setPritority(ex,5)로 된다. 새로운 메소드는 우선순위를 정수형 매개변수로 취하여 상위 한계치와 비교한다. 매개변수의 값이 상위한계치보다 크면 매개변수의 값이 상위한계치가 된다.

### 3.2.1 메소드 참조 수정

클래스 또는 클래스 인스턴스의 메소드는 constant pool entry에 CONSTANT_Methodref(●)로서 표현되어진다.

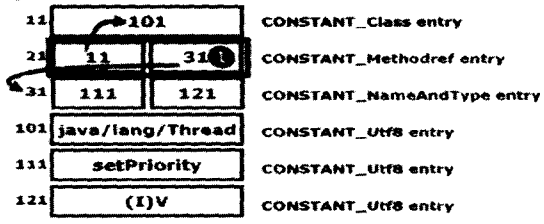


그림 4 Thread.setPriority(I)V에 대한 참조

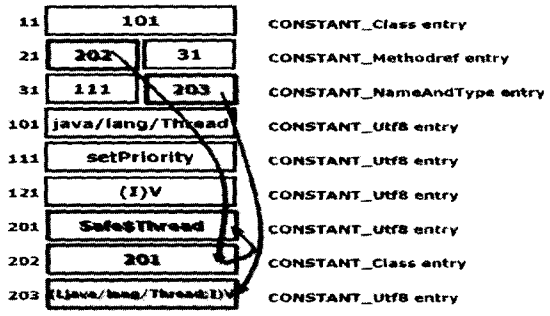


그림 5 Safe$Thread.setPriority(Ljava/lang/Thread;I)V로의 참조

이는 결과적으로 그림 4에서 java/lang/Thread 참조하던 것이 그림 5와같이 Safe$Thread를 참조하도록 수정된다.

### 3.2.2 메소드 호출 명령 수정(Method Invoking Instruction Modification)

표2를 통하여 메소드 발생을 구현하는 다양한 자바가상기계 명령중에서 2가지를 살펴본다[9].

**표 2) 명령(Instruction)의 설명**

| | |
|---|---|
| *invokevirtual* | 클래스의 인스턴스의 메소드의 발생 |
| *invokestatic* | static 클래스의 메소드 발생 |

클래스의 메소드 호출은 단순히 피연산자 스택으로 푸쉬되는 매개변수만이 필요하다. 기존의 메소드는 *invokevirtual*에 의해 피연산자 스택에 푸쉬되었으나 수정된 메소드의 경우 *invokestatic*에 의해 수행되도록 바이트코드가 바뀌어지며 그 결과 메소드가 수정되어 *Safe$Thread.SetPriority(Ljava/lang/Thread;I)V*로 된다. staic 클래스처럼 수정되었으므로 상속이 가능하게 된다. 메소드 수준의 수정은 바이트코드 명령과 constant pool의 수정으로 가능하다. 클래스 수준 수정은 *final* 클래스 또는 인터페이스에는 적용될 수 없으므로 메소드 수준의 수정이 이루어 져야 한다.

## 4. 결론

본 논문에서는 애플릿의 유해한 행동을 소스코드로 역컴파일(decompile)한 후 유형을 판단하는 방법을 사용하는 것이 아니라, 'Java Virtual Machine Specification[4]'에서 제시하는 클래스 파일의 형식을 이용하여 바이트코드 상태에서 클래스수준 수정을 수행하여 유해한 행동을 사전에 제한하는 바이트 코드 수정을 통한 자바 애플릿 보안기술에 대해 살펴보았다. 자바 바이트코드 수정중 클래스 수준 수정은 자바의 상속성을 이용하였으므로 final 클래스와 인터페이스를 사용하여 구현(*implement*)한 객체와 같이 상속할 수 없는 경우에 대해서는 적용이 불가능하다. 그 해결책으로는 바이트코드 명령과 Constant pool을 수정하는 바이트코드 수준 수정이 있다. 바이트코드 수준 수정을 이용하면 서버, 클라이언트, 그리고 브라우저에 대해 어떠한 별도의 작업도 필요하지 않다. 다만, 프락시 서버에서 사전에 정의한 유해한 행동을 제어하는 Safe 클래스에 의해 수정된 후 브라우저에 보여진다.

자바로 구현된 Muffin 프록시 서버를 사용한 관계로 속도측면에서 상당히 부하가 작용된다. 그리고, Safe 클래스가 온라인으로 패턴이 추가되지 않고, 직접 유형별로 추가해야 한다는 단점이 있다.

지금까지 살펴본 내용을 토대로 프록시서버를 이용하여 Safe 클래스가 구현되어야 하며 또한 패턴을 기초로 한 유형별 바이트코드 수정이 되어야 한다. 자바언어에서의 문제점이지만 바이트코드 수정시간이 과다하지 않도록 핫스팟(HotSpot)등을 통한 수행속도 향상 작업이 필요하다.

## 5. 참고 문헌

[1] Dirk Balfanz and Edward W.Felten, **A Java Filter**, *Technical Report 97-567*, Department of Computer Science, Princeton University, 1997.

[2] Drew Dean, Edward W.Felten and Dan S.Wallach, **Java Security : From Hotjava to Netscape and beyond**, In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996.

[3] Edward W.Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach, **Web spoofing : An Internet Con Game**, *Technical Report 540-06*, Department of Computer Science, Princeton University, Feb 1997.

[4] Tim Lindholm and Frank Yellin, **The Java Virtua Machine Specification 2nd Edition**, http://java.sun.com docs/books/vmspec/index.html

[5] Gary McGraw and Edward W. Felten, **Java Security : Hostile Applets, Holes, and Antidotes**, John Willey & Sons, 1997.

[6] David M, Martin Jr, Sivaramakrishnan Rajagopalan, and Aviel D. Rubin. Blocking **Java Applets at the Firewall**, In *Proceedings of the 1997 Internet Society Symposium on Network and Distributed System Security*, Feb 1997.

[7] Gary McGraw, Edward W. Felten, **SecuringJava Getting Down to Business with Mobile Co** http://www.securingjava.com, 1999.

[8] Markus Dahm. **Byte Code Engineeri** http://www.inf.fu -berlin.de/~dahm/JavaClass/index.html

[9] Bill Venners. **How the Java Virtual machine hand method invocation and return**, http://www.javawo com/javaworld/jw-06-1997/jw-06-hood_p.html

[10] Inshik Shin, John C. Mitchell, **Java Bytec Modification and Applet Secur** http://theory.stanford.edu/~vganesh/project.html